Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

DA256B, DT256A: Algorithms and Data Structures

Kamilla Klonowska

Seminar 3: Hashing, Trees

Goals

The goal of this seminar is to prepare the students to fulfill the learning outcomes

Knowledge and understanding

After completing the course students must

- be able to explain the algorithm theory and data structures on a general level (1)
- be able to explain the qualitative and quantitative methods applicable to research issues in computer science (2)

Skills and abilities

- be able to apply the algorithm theory and data structures in practice (3)
- be able to describe and discuss its own expertise with various student groups (5)
- be able to work independently and in groups (6)

Judgement and approach

- be able to reflect, motivate and argue for knowledge in algorithm theory and data structures (8)

The students train: *Personal and professional skills and attributes*: Analytical reasoning and problem solving, as well as *Interpersonal skills*: teamwork and communication. (CDIO goals).

Before the Seminar –

The seminar is mandatory. A student that did not upload the tasks is not allow to participate in the seminar.

All four tasks must be solved and they should be solved individually.

It is not sufficient to only solve the tasks, the student must also be able to explain the source code and explain how each algorithm and/or logic behind the solution otherwise the tasks will not be approved.

Note that hints of how some of the different algorithms work or how to implement them can be found in the course literature [1] but you should make sure to understand the algorithms before implementing them.

Remember having your laptop with you in the seminar!!

Tasks 1 ([1], Ch.6)

Implement in Java a binary heap using the following example:

a. Present the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, 2 one at a time, into an initially empty binary heap (**algorithm 1**).

b. Present the result of using the linear-time algorithm to build a binary heap using the same input (**algorithm 2**).

The graphical presentation of the results is presented below (Fig.1). Present your results e.g. in an array.

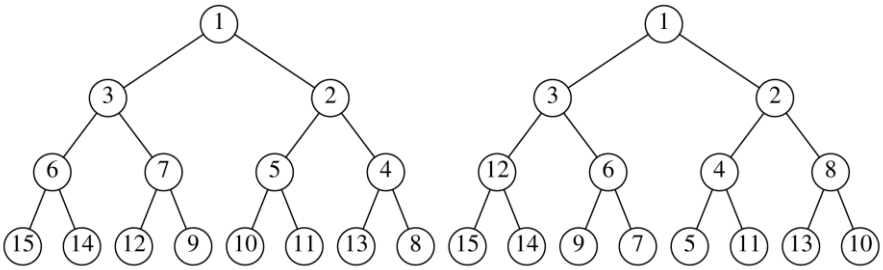

Fig. 1. Output of **algorithm 1** - task 1a (left tree) and **algorithm 2** - task 1b (right tree).

c. Present the results of **traversing** both trees using all four traversing strategies: in-order, pre-order, post-order, level-order.

   Note: some of the routines/code you will find in a book or lecture slides

d. Measure the complexity of **both algorithms** by changing the input, e.g. by using the numbers (100, 1000, 10000, etc) from the file from the Seminar 1.

e. Investigate in priority queues, which is more expensive operation: deleting the minimum element, or the insertion by providing these operations on both trees.

Note: Be creative! The presentation must be representative.

Tasks 2

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x mod 10, implement the following to present the results of:

a) Separate chaining hash table.

b) Hash table using linear probing.

c) Hash table using quadratic probing.

d) Discuss the complexity

e) Hash table with second hash function $h_2(x) = 7 - (x \bmod 7)$.

f) Rehash the hash table from (e) to a table with a double size.

g) Discuss other rehashing functions

Note: Be creative! Do the notes so you take it to the seminar to lead a discussion.

Task 3 - Word Puzzle (ex. 5.18, [1])

Implement the word puzzle program using the algorithm described below.

Note: We can get a big speed increase by storing, in addition to each word W, all of W's prefixes. (If one of W's prefixes is another word in the dictionary, it is stored as a real word.) Although this may seem to increase the size of the hash table drastically, it does not, because many words have the same prefixes. When a scan is performed in a particular direction, if the word that is looked up is not even in the hash table as a prefix, then the scan in that direction can be terminated early. Use this idea to write an improved program to solve the word puzzle.

	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t

Fig. 2. Sample word puzzle (p.22, [1])

The Algorithms:

The input consists of a two dimensional array of letters and a list of words. The object is to find the words in the puzzle. These words may be horizontal, vertical, or diagonal in any direction. As an example, the puzzle shown in Fig. 2 contains the words *this*, *two*, *fat*, and *that*. The word *this* begins at row 1, column 1, or (1,1), and extends to (1,4); *two* goes from (1,1) to (3,1); *fat* goes from (4,1) to (2,3); and *that* goes from (4,4) to (1,1).

There are at least two straightforward algorithms that solve the problem. For each word in the word list, we check each ordered triple (*row, column, orientation*) for the presence of the word. This amounts to lots of nested for loops but is basically straightforward.

Alternatively, for each ordered quadruple (*row, column, orientation, number of characters*) that doesn't run off an end of the puzzle, we can test whether the word indicated is in the word list. Again, this amounts to lots of nested **for** loops. It is possible to save some time if the maximum number of characters in any word is known. (p.22, [1])

Running time analysis:

If the second algorithm is used, and we assume that the maximum word size is some small constant, then the time to read in the dictionary containing W words and put it in a hash table is O(W). This time is likely to be dominated by the disk I/O and not the hashing routines. The rest of the algorithm would test for the presence of a word for each ordered quadruple (*row, column, orientation, number of characters*). As each lookup would be O(1), and there are only a constant number of orientations (8) and characters per word, the running time of this phase would be O(R·C). The total running time would be O(R·C+W), which is a distinct improvement over the original O(R·C·W). We could make further optimizations, which would decrease the running time in practice; these are described in the exercises. (p.238, [1])

Tasks 4 - Optional ([1], Ch.12)

Implement in Java the following exercise

a. Compare the time complexity of the following tree structures:
    a. Binary Search Tree,
    b. Heap,

c. AVL,

d. Red-Black Tree.

Find the method that you think is the most suitable for this problem. Of course, motivate your choice. Present the results in a table and a diagram.

b. In what kind of problems you'd want to use/implement each of the tree structure?

Note: Be creative! The presentation must be representative.

During the Seminar –

Each student is responsible for one (random) Task for the examination. The student is responsible for the task, i.e. explains the solution, presents a code, discusses the complexity / efficiency of the code/algorithm, discusses other students' solutions, as well as is responsible for the analysis of the results of the current task.

References

1. Weiss, Mark. A. (2012), Data structures and algorithm analysis in Java. 3rd edition Harlow, Essex : Pearson. (632 p).